val Research Laboratory
hington, DC 20375-5000

NRL Report 9267

Parallel Access Main Memory (PAMM) User's Manual, Version 1.0

TODD J. ROSENAU

Integrated Warfare Technology Branch
Information Technology Division

MONA EL-KADI

Locus, Inc.
Alexandria, VA 22303

September 15, 1990

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final 6/87 — 3/90

4. TITLE AND SUBTITLE

Parallel Access Main Memory (PAMM)
User's Manual, Version 1.0

5. FUNDING NUMBERS

PE - 63223C
PN - 55-2354-C-0
WU - DN155-502

6. AUTHOR(S)

T. J. Rosenau and Mona El-Kadi*

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Washington, DC 20375-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL Report 9267

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Strategic Defense Initiative Organization
Washington, DC 20301-7100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

*Locus, Inc, Alexandria, VA 22303

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Here we describe the structure and use of the parallel access main memory (PAMM) database management system (DBMS) interface for C programmers. PAMM is a high-speed, high-throughput DBMS that allows concurrent access to data stored in a distributed data structure across all of the processors in a general-purpose, multiprocessor computer. Concurrency is achieved by allowing several tasks executing on different processors to access the database at the same time with a minimum of locking. This allows an application's tasks to proceed with a minimum of delay. Its high speed is achieved by keeping most of the data and index structures resident in main memory, thus reducing the number of disk accesses performed. The data structure and storage method for the DBMS are a multidirectory hashing, distributed lock system that Sakti Pramanik and Charles Severance at Michigan State University developed for the Naval Research Laboratory (NRL). PAMM has been implemented on a 32-processor BBN Butterfly GP1000 at NRL.

14. SUBJECT TERMS

Parallel processing
Hashing
Database management systems

15. NUMBER OF PAGES

42

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

CONTENTS

PARALLEL ACCESS MAIN MEMORY (PAMM)

User's Manual, Version 1.0

1.0 INTRODUCTION

Often, when a data-intensive application is running, the need to access all of the data efficiently and at random can be the most time-consuming portion of the execution time. This can happen when the computer application is running or has an insufficient amount of main memory or slow secondary memory. A conventional database management system (DBMS) running on a typical computer must also share resources with the application for which it is storing data and any other applications that may be running that may seriously degrade its performance. What is needed is a DBMS with an abundance of main memory that has enough computing power to keep all of the memory busy in which to store data[1].

One computer that satisfies these requirements is BBN's Butterfly GP1000 [2,3]. This computer can have up to 128 processors with a maximum of 512 Mbytes of main memory. Each processor is a Motorola 68020, with 4 Mbytes of main memory. The Naval Research Laboratory (NRL) has developed a high-speed, concurrent-access, main memory DBMS that has been implemented on a 32-processor GP1000. Here we describe the interface to the parallel access main memory (PAMM) DBMS on the GP1000.

PAMM is based on research by Sakti Pramanik and Charles Severance at Michigan State University. They developed multidirectory hashing [4], which is a data-hashing scheme involving multiple hash tables instead of just one. This hashing method was then modified [5-8] to allow concurrent access to the multiple hash tables by multiple processors across distributed, shared memory. The PAMM DBMS is a working prototype that uses these algorithms; it can be used as the data manager for an application running on the Butterfly GP1000.

2.0 SYSTEM OVERVIEW

PAMM was developed as a high-speed, high-throughput, concurrent-access DBMS for use on a general-purpose parallel processor such as the GP1000 [9]. It maintains data records in a dynamic, distributed data structure [10] that allows concurrent access to records within the PAMM structure with a minimum of locking. Access to data is through a two-phase hashing function to distributed hash tables (Fig. 1). The first phase of the hash function determines the hash table or *directory* in which the data record must be placed and the second phase determines where in this directory the record must go. The number of directories is much greater than the number of processors used, so each processor has several different directories resident within its memory. Once the directory is determined, the record will be hashed into a chain-linked hash table in that directory. Thus, to determine where a record will be located, two separate hash function computations must be performed. If a maximum limit can be placed on the time to search a chain, then the access time to insert, find, update, or delete a record should be upper bounded by some constant.

Manuscript approved May 22, 1990.

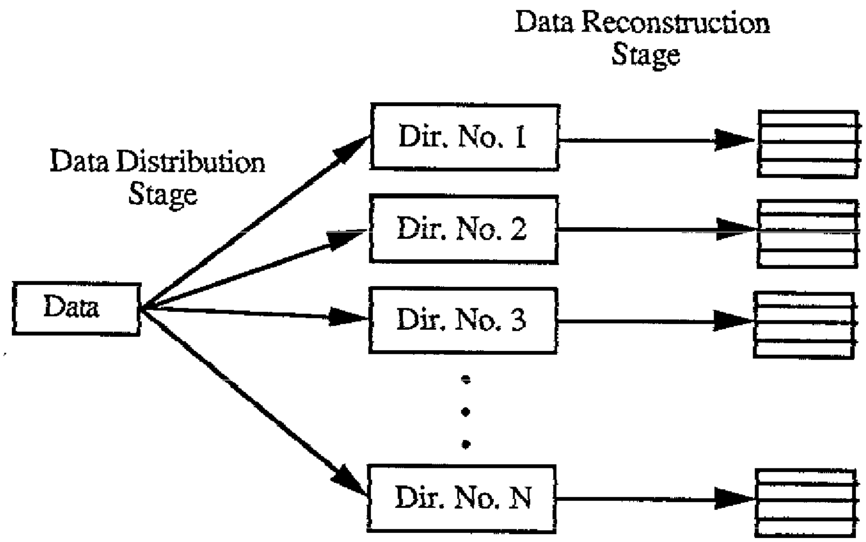

Fig. 1 - Two-phase hashing method

This method works fine until the number of records within a single directory increases to the point where searching the linked list of records becomes a major component of the time to find a record. Once this point is reached, performance begins to degrade unless the size of each directory is reduced. At a user-specified threshold, PAMM creates a new directory, and the records from the overfilled directory are redistributed between the old and the new directories. Statistics for each directory are kept so that memory use and average chain length can be monitored. Thus, PAMM is a dynamic, concurrent-access DBMS with a user definable upper bound on the maximum access time.

Since PAMM is a usable DBMS, it has long-term storage mechanisms for maintaining table structure and table data over multiple sessions. Long-term storage is facilitated by using data dictionaries to store information about tables and their attributes. When the database is open (loaded into main memory), the dictionaries and tables are loaded into main memory as PAMM structures; otherwise, they are stored as Mach 1000 (UNIX) files. Two dictionaries are associated with each database, where each database is a separate collection of tables and data and has a unique name. The main dictionary is the *table dictionary*, which stores necessary, long-term information about each table. The other dictionary is the *attribute dictionary*, which stores information about every attribute in each table. Since each dictionary is also a table, each is described in both dictionaries. The table and attribute dictionaries are named "*db_name*.td" and "*db_name*.ad," respectively, where "*db_name*" is the name of the current database. Besides the file extensions ".td" and ".ad" for data dictionaries, table names end in ".dat" and table schema files end with ".scm". These are described in more detail.

2.1 Schema Files

For the database to create new tables (including the dictionaries), information about the structure of the new table must be passed into the system. One method of accomplishing this task is to pass a list or data structure of table format parameters to the create table routine. This is awkward and difficult to code into an application program. Instead we have chosen to associate a *schema file* with every table and data dictionary within a database. A schema file is a separate file associated with a table and contains instructions specifying the exact structure of the respective table. This method allows all table formats to be viewed and modified without having to change any code. Schema files have a uniform, simple layout for easy, fast creation or modification of a table's format. The syntax of a schema file is as follows:

```
database db_name
     attr attribute1_name,     type field_type,  key 1
     attr attribute2_name,     type field_type,  key 0
     attr attribute3_name,     type field_type,  key 0
                .                     .             .
                .                     .             .
                .                     .             .
     attr attributeN_name,     type field_type,  key 0
end
```

A schema file consists of several lines of text describing various features of the table. In the example above, all necessary keywords are shown in **bold** including commas. The first line in a schema file specifies the name of the database in which this file belongs. The keyword **database** must be the first word in the schema file, followed by the name of the database to which this table belongs. Following this line are the descriptions of all attributes within this table. For clarity, each line describes only one attribute. The first word in an attribute description line is the keyword **attr**, followed by the name of the attribute. The only restrictions on an attribute's name are that it can be no longer than 31 characters and cannot contain a comma. The attribute's name is then followed by a comma and the keyword **type**. After the keyword, the type of the attribute is specified, followed by another comma. The last section in the attribute description is signaled by the keyword **key**, followed by a number specifying whether or not the attribute is a key field. If the number is a zero, the attribute is not a key field. Otherwise, the integer value represents the ranking of the attribute in the key (1 is the primary field in the key, 2 is the secondary field in the key, etc.). Unfortunately, at this point in development, the only valid values for the key priorities are zero and one, and the key field must be an integer.

TYPE	C TYPE	SIZE	DESCRIPTION
char	char	1 byte	8-bit character
short	short	2 bytes	16-bit short integer
integer	int	4 bytes	32-bit integer
long	long	4 bytes	32-bit integer
float	float	4 bytes	32-bit, single-precision, floating point
double	double	8 bytes	64-bit, double-precision, floating point
string *N*	char[N]	N bytes	N–1 byte character string, plus null terminator

The valid types for an attribute are **char**, **short**, **integer**, **long**, **float**, **double**, and **string *N***. The type **char** (C type: char) is for storing a single character (or other 8-bit field), and occupies one byte of memory. The type **short** (C type: short) is a 16-bit integer that occupies two bytes of memory. The type **integer** (C type: int) is a 32-bit integer field that occupies four bytes of memory. The type **long** (C type: long) is also a 32-bit integer field that occupies four bytes of memory. Although these two fields are the same size, the provision is made to handle both data types for those applications that use both, or PAMM may be ported to a multiprocessor that does. The next field is **float** (C type: float), a single-precision, floating-point variable of length 32-bits (four bytes). The type **double** (C type: double) is a double precision, floating-point variable of length 64-bits (8 bytes). Lastly, the type **string** *N* (C type: char[]) is a character string of length *N*, including a null terminator.

Both data dictionaries are stored as tables and thus have corresponding schema files. The table dictionary's schema file is listed below as an example and provides a definition of its attributes:

table_dict.scm — table dictionary schema file

```
database master
     attr table_id,       type long,        key 1
     attr table_name,     type string 32,   key 0
     attr num_attrs,      type short,       key 0
     attr first_attr,     type long,        key 0
     attr rec_length,     type short,       key 0
     attr max_chains,     type integer,     key 0
     attr split_val,      type float,       key 0
     attr overflow,       type float,       key 0
     attr initial_m,      type integer,     key 0
     attr max_dirs,       type integer,     key 0
     attr sup_dups,       type short,       key 0
     attr sort_chain,     type short,       key 0
end
```

The database name associated with both dictionaries is **master**, which is ignored when the dictionaries are created. Instead, the database name is assigned when the application creates a new database and is stored in the name of both dictionaries (i.e., *db_name*.td and *db_name*.ad, where *db_name* is the name of the database). Since the table and attribute dictionary schema files are the same for all databases, they are named "table_dict.scm" and "attr_dict.scm," respectively.

The first attribute of the table dictionary is the key field named *table_id* of type **long**. At this stage in PAMM's development, the only key field allowed is a 32-bit integer value that must be the first four bytes in every record. The second attribute is *table_name*, which is of type **string 32**. Because of internal PAMM design decisions, database, table, and attribute names can be a maximum of MAX_NAME (32) characters, including file extensions and null terminators. Thus, database names can be 28 characters plus extension (".td\0" or ".ad\0"); table names can be a maximum of 27 characters plus extension (".dat\0"); and attributes can be a maximum of 31 characters plus null terminator ("\0"). The third attribute is *num_attrs* of type **short**. This field contains the number of attributes in the table being described. The fourth attribute is *first_attr*, of type **long**. It contains the attribute ID for the first attribute in the attribute dictionary. The attribute IDs for a particular table are sequentially increasing integers ranging from *first_attr* to (*first_attr* + *num_attrs* -1), and are disjoint from any other tables' attribute IDs. The fifth attribute is *rec_length* of type **short** — the length of the table's record in bytes. The sixth attribute is *max_chains* of type **integer**, and specifies the maximum number of linked lists or chains allowed in each directory. The seventh attribute is *split_val* of type **float**; this floating point number specifies the splitting threshold for a *P* directory. If the current directory is number *P* and the average chain length is greater than or equal to *split_val*, this directory can be split in two. Average chain length is computed as follows:

$$\frac{\text{number of records in directory}}{\text{number of nonempty chains in directory}}$$

The eighth attribute is *overflow* of type **float**. It also specifies a splitting threshold for a directory that is not the *P* directory. The ninth attribute is *initial_m* of type **integer**. When a table is first loaded into memory, a default number of empty directories can be created — a number set by the user ahead of time. The tenth attribute is *max_dirs* of type **integer**; this specifies the maximum number of directories allowed for a particular table. The eleventh attribute is *sup_dups* of type **short**; this is a Boolean value, which is set to TRUE if duplicate records are to be suppressed and FALSE if duplicates are allowed. The twelfth and last attribute is *sort_chain* of type **short**; this is also a Boolean value, which is TRUE if the chains in each directory are to be kept sorted and FALSE if the chains are not sorted.

Another dictionary schema file is listed for the attribute dictionary, "attr_dict.scm:"

attr_dict.scm — attribute dictionary schema file

```
database master
     attr attr_id,       type long,          key 1
     attr attr_name,     type string 32,     key 0
     attr table_id,      type long,          key 0
     attr attr_type,     type short,         key 0
     attr attr_pos,      type short,         key 0
     attr attr_offset,   type short,         key 0
     attr attr_length,   type short,         key 0
     attr attr_priority, type short,         key 0
end
```

The first attribute is *attr_id* of type **long**; it is the attribute ID for this particular attribute. A unique, long integer ID exists for all attributes in the tables. The second attribute is *attr_name* of type **string 32**; it is a character string of length 32 characters. With the null terminator ("\0"), the maximum length of an attribute name is 31 characters. The third attribute is *table_id* of type **long**; this is the table ID for the table to which the attribute belongs. The fourth attribute is *attr_type* of type **short**, which specifies the data type of the attribute. The set of possible types are: {**char, short, integer, long, float, double, string** *N*}. The fifth attribute is *attr_pos* of type **short**; this specifies the position of the attribute within the record with the first attribute's position set to one. The sixth attribute is *attr_offset* of type **short**; this specifies the byte offset of the attribute within the record relative to the first attribute in the record. The byte offset of the first attribute in a record is always zero. The seventh attribute is *attr_length* of type **short**; this number stores the size of the attribute in bytes . The eighth and last attribute is *key_priority* of type **short**; this number determines whether or not the attribute is a part of the key. If this value is zero, it is not a part of the key; otherwise, the value will determine the attribute's ranking within the key (i.e., one is the major component of the key field, two is the second major component of the key, etc.. At this point in PAMM's development, the key field can consist of only one attribute, and it must be an integer field in the first position in the record.

3.0 INTERFACE DESCRIPTION

PAMM is implemented as a library of routines that is compiled with the application program. It was written in the C programing language. A simple interface accesses the underlying data structures. The interface routines to PAMM are divided into four distinct groups: initialization, database manipulation, table manipulation, and record manipulation. The initialization routine creates all necessary global data structures and makes them available to all of the future child tasks. Database manipulation routines allow a database to be created, opened, closed and destroyed. Table routines allow specified tables to be created, opened, loaded, unloaded, saved, closed, and destroyed. Record routines allow data records to be inserted, read, updated, and deleted.

Application programs that use the PAMM DBMS must contain the include file "**pamm.h**" in all source code modules. For all applications, the command "#define MAIN" **must** precede the command "#include <pamm.h>" in one and only one source code module (preferably the module with the main routine declared in it). If the application has more than one source code module, the remaining modules **must** call the command "#undef MAIN" before calling the command "#include <pamm.h>". All interface routines return an integer value displaying the status of the returning routine that can be found in the **pamm.h** include file. The interface routines are described below.

3.1 Initialization Routine

P_init() — This function allocates and initializes all of the global data structures that PAMM uses. It must be called before any other PAMM routine except *P_create_DB()*, *P_destroy_DB()*, *P_create()* and *P_delete_table()*, and it must be called by the parent task of all tasks that will access the PAMM structures, because this function will set the memory inheritance for all global memory. It returns the values OK if *P_init()* completed successfully or NOT_OK if an error occurred.

3.2 Database Manipulation

P_create_DB("*DB_name*") — This function verifies that a database with the name "*DB_name*" does not already exist and then makes a new database by creating a data dictionary and an attribute dictionary named "*DB_name*.td" and "*DB_name*.ad," respectively. It returns the values OK if *P_create_DB()* completed successfully, DB_ALREADY_EXISTED if there previously existed a database with the same name, or DB_CREATE_FAILED if an error occurred.

P_open_DB("*DB_name*") — This function opens the database named "*DB_name*" and loads the two dictionaries into PAMM tables. *P_open_DB()* must be called before any of the table and record manipulation routines, but after *P_init()*. It returns the values OK if *P_open_DB()* completed successfully, DB_DICTIONARY_DAMAGED if one of the dictionaries became corrupted, or DB_OPEN_FAILED if an error occurred.

P_close_DB("*DB_name*", *wait*) — This function closes the database named "*DB_name*." It takes two parameters: *DB_name* and *wait*. "*DB_name*" is the database to be closed and *wait* is of type BOOLEAN that specifies whether *P_close_DB()* should wait for all tables to close before closing the database, or, if any tables are still open, return immediately with a code. *P_close_DB()* unloads all tables still residing in main memory in PAMM tables into secondary storage. Finally, it closes and unloads the two dictionary tables. The routine *P_close_DB()* returns the values OK if it completed successfully, OPEN_TABLE_BUT_CANT_WAIT if tables are still open but the wait variable was not set, or DB_CLOSE_FAILED if an error occurred.

P_destroy_DB("*DB_name*") — This function destroys the two dictionaries and all tables belonging to the database named "*DB_name*" by physically removing the Mach (UNIX) files from disk; thus this operation is nonreversible. The database to be destroyed must not be open or it will fail. *P_destroy_DB()* returns the values OK if it completed successfully, DB_DID_NOT_EXIST if the database to be destroyed did not exist, or DESTROY_DB_FAILED if an error occurred.

3.3 Table Manipulation Routines

P_create("*table_name*") — This function creates a new table called "*table_name*.dat" according to the description in the new table's schema file. Every table has a corresponding schema file, including both dictionaries (see Section 2.1) that tells which database the table belongs in and the names and descriptions of all the attributes in the table. This routine must be called before *P_open_DB()* because it will add the new table's description directly into the two dictionary files in secondary storage — not into the dictionaries' PAMM structures in main memory. *P_create()* returns the values OK if the routine is completed successfully, SCHEMA_NOT_FOUND if the table's schema file was not present, SCHEMA_SYNTAX_ERROR if the schema file description had an error, TABLE_ALREADY_EXISTS if that table already existed, or CREATE_FAILED if an error occurred.

P_load("*table_name*", *&new_parameters*) — This function loads the records from the file "*table_name*.dat" into a PAMM table in main memory. The table's database must already be open, and the table must already have been created. *P_load()* must be called before the table can be opened (*P_open()*) for access even if there are no records to be loaded, because *P_load()* creates the new PAMM data structures in main memory. If it is desired that the PAMM table parameters be changed from the default values, *new_parameters* will be passed in with the new set of parameters. *New_parameters* is of PARM_TYPE, which can be found in **pamm.h**, along with the default values stored in the variable, *initial_table*. *P_load()* returns OK if it completed successfully, TABLE_NOT_EXIST if that table was not listed in the table dictionary, TOO_MANY_TABLES if there are too many loaded tables, (MAX_OPEN_TABLE = 20) or LOAD_FAILED if an error occurred.

P_open("*table_name*", *access_mode*, *&td*) — This function opens the table called "*table_name*." The table must already be loaded before it can be opened. The variable *td* must be declared as a record of type PAMM_TYPE in the calling program, and its address is passed into *P_open()*. Upon returning from *P_open()*, *td* is a valid table descriptor. The table is opened with the permission level given by *access_mode*. Valid access types are READ, WRITE, and READ_WRITE. *P_open()* returns OK if the table was opened successfully, TABLE_NOT_EXIST if the table was not found in the table dictionary, TABLE_NOT_LOADED if the table was not loaded into main memory, or INVALID_ACCESS_TYPE if it was opened with an incorrect access type.

P_close(*&td*) — This function closes the table referenced by the table descriptor *td* and invalidates the table descriptor when it returns. *P_close()* returns OK if the table was closed successfully or TABLE_NOT_OPEN if the table was not previously open.

P_save("*table_name*", "*disk_file_name*") — This function unloads the data in main memory from the table named *table_name* into a file on disk called "*disk_file_name*"; it does not remove the table from the main memory database, but it makes a copy of the table onto disk. This routine saves the contents of a table without having to close the table. We recommend that "*disk_file_name*" not equal "*table_name*.dat" because the intermediate file will be overwritten when the table is unloaded. *P_save()* returns OK if it completed successfully, TABLE_NOT_EXIST if *table_name* is not in the database, or SAVE_FAILED if an error occurred.

P_unload("*table_name*") — This function unloads the data in main memory from the table named "*table_name*" into the file on disk, "*table_name*.dat," and then removes the table from main memory. A table that is open cannot be unloaded. *P_unload()* will return OK if it completed successfully, TABLE_NOT_EXIST if "*table_name*" is not in the database, TABLE_STILL_OPEN if "*table_name*" is still open, or UNLOAD_FAILED if an error occurred.

P_delete_table("*table_name*") — This function deletes a table with the name "*table_name*.dat." It requires that a schema file of the correct format exists called "*table_name*.scm." *P_delete_table()* deletes the given table's information from both data dictionaries without loading the database into main memory and also deletes the data file for the table, which is called "*table_name*.dat." Thus *the P_delete_table()* must be called before *P_open_DB()* or after *P_close_DB()*. *P_delete_table()* returns OK if the table was deleted successfully from the database, SCHEMA_NOT_FOUND if the schema file named "*table_name*.scm" was not found, SCHEMA_SYNTAX_ERROR if the schema file "*table_name*.scm" was not in the correct form, TABLE_NOT_EXIST if the table was not in the table dictionary, or DELETE_TABLE_FAILED if an error occurred.

3.4 Record Manipulation Routines

P_insert(*&td, &data_rec*) — This function inserts a record pointed to by *data_rec* into the table described by *td*. The key for the record is assumed to be the first four bytes of *data_rec*. *P_insert()* returns OK if the record was inserted successfully, NO_WRITE_PERM if the table was not opened with WRITE or READ_WRITE permission, or DUPLICATE_KEY if a record with that key is already in the table and duplicate suppression is turned off.

P_read(*&td*, *key*, *&data_rec*) — This function reads a record from the table described by *td* with the key, and *key* into a buffer pointed to by *data_rec*. *Data_rec* must be the address of a buffer that exists in the calling program's memory. The table must be open for READ or READ_WRITE permission. *P_read()* returns OK if the record was read successfully, TABLE_NOT_OPEN if the table was not open or the permissions were wrong, or RECORD_NOT_FOUND if the record with key, *key*, is not in the table.

P_update(*&td, &data_rec*) — This function replaces a record in the table described by *td* with the contents of *data_rec*. The key for the record is assumed to be the first four bytes of *data_rec*. *P_update()* returns OK if it completed successfully, NO_WRITE_PERM if the table was not opened with WRITE or READ_WRITE permission, or RECORD_NOT_FOUND if the record with the specified key was not found.

P_delete(*&td*, *key*) — This function deletes a record with key, *key*, from the table described by *td*. *P_delete()* returns OK if the record was deleted successfully, NO_WRITE_PERM if the table was not opened with WRITE or READ_WRITE permission, or RECORD_NOT_FOUND if a record with the specified key is not found.

4.0 PROGRAMMING EXAMPLES

All PAMM interface routines return an integer status variable that can be found in the include file **pamm.h**. For the following examples, we use an integer variable named *result* for storing all status values.

4.1 PAMM Initialization

Before PAMM can be used, it must be initialized by calling *P_init()*; this allocates and initializes all global data structures that PAMM needs. Thus, *P_init()* must be called before any routine that accesses the PAMM tables (all routines except *P_create_DB()*, *P_destroy_DB()*, *P_create()*, and *P_delete_table()*). *P_init()* must also be called in the task that is a common ancestor to all tasks that access the PAMM tables because only those tasks that are its children inherit its memory address space. The code for initializing PAMM in an application program is:

```
if ((result = P_init()) != OK)
{
     printf("ERROR P_init returned %d\n", result);
}
```

4.2 Creating a Database

To create a database, two files must exist in the directory of the database, "attr_dict.scm" and "table_dict.scm." These two files come with the PAMM package and can be moved or copied but must never be altered. The ".scm" suffix on a file name means that the file is a schema file and that it specifies the structure of a table, such as the number, type, and order of the attributes (see Section 2.1). The command *P_create_DB(database_name)* creates the attribute and table dictionaries for the database name passed in the parameter *database_name* according to the two schema files. The code for creating a new database is:

```
char database_name[29];

strcpy(database_name, "DB_name");
if ((result = P_create_DB(database_name)) != OK)
{
     printf("ERROR P_create_DB returned %d\n", result);
}
```

where "DB_name" can be any alphabetic character string no longer than 28 characters. After this code is run, the files "DB_name.td" and "DB_name.ad" will exist in the directory, and tables may then be put in the database.

4.3 Opening a Database

To open a database, the command *P_open_DB(database_name)* must be called. The parameter *database_name* specifies the name of the database to be opened. As before, the maximum length of the name of the database is 28 characters (excluding file extension). This operation allocates and initializes all global data structures that PAMM needs. *P_open_DB()* then loads into PAMM tables both data dictionaries (named "DB_name.td" and "DB_name.ad"), where "DB_name" is the name of our example database. The code for opening a database is:

```
char database_name[29];

strcpy(database_name, "DB_name");
if ((result = P_open_DB(database_name)) != OK)
{
     printf("ERROR P_open_DB returned %d\n", result);
}
```

4.4 Closing a Database

To close a database, the command *P_close_DB(database_name, wait)* must be executed. This command unloads all tables currently in main memory back to secondary memory that belong to the database specified by the parameter *database_name*. The *wait* parameter specifies if *P_close_DB()* should wait for any open tables to be closed (*wait* = TRUE) or if it should return immediately without closing the rest of the database (*wait* = FALSE). The code for closing a database is:

```
int  wait;
char database_name[29];

wait = FALSE;
strcpy(database_name, "DB_name");
if ((result = P_close_DB(database_name, wait)) != OK)
{
     printf("ERROR P_close_DB returned %d\n", result);
}
```

4.5 Destroying a Database

To destroy a database and all of its tables, the command is *P_destroy_DB(database_name)*. This command deletes all the .dat files of the tables in the database whose names are specified by the parameter *database_name* and also deletes the two data dictionaries, "*database_name*.ad" and "*database_name*.td." All of the tables and the database must be closed before executing. The code for destroying a database is:

```
char database_name[29];

strcpy(database_name, "DB_name");
if ((result = P_destroy_DB(database_name)) != OK)
{
     printf("ERROR P_destroy_DB returned %d\n", result);
}
```

4.6 Creating a Table

To create a new table, the command is *P_create(table_name)*. A table is created in the same directory as the database to which it will belong, and its name will be specified in the parameter *table_name*. The table name must be no longer than 27 characters, excluding the file extension, ".dat\0." This command searches in the same directory for a schema file named "*table_name*.scm," which must be in the correct format (see Section 2.1 for format details). *P_create()* must be called before *P_open_DB()* or after *P_close_DB()* because it places the information about the table and its attributes directly into the two data dictionary files, not into a PAMM structure. The code to create a new table is:

```
char table_name[28];

strcpy(table_name, "table1");
if ((result = P_create(table_name)) != OK)
{
     printf("ERROR P_create returned %d\n", result);
}
```

4.7 Loading a Table

When a database is opened, only the two data dictionaries are loaded into the main memory data structure. It is up to the application to load into memory whatever database tables it wants to use. Loading a table into memory is done with the *P_load(table_name, &new_parm)* function, which takes two parameters — *table_name* and *new_parm*. The name of the table to be loaded is passed in the character string variable *table_name*. The parameter *new_parm* is of type PARM_TYPE, which is defined in **pamm.h**. It lists the parameters for the table that can be adjusted at load time to alter database performance. If no changes are desired, the value NULL should be passed in *new_parm*'s position. If the parameters are changed, they replace the default values for all subsequent table loadings until they are changed again. The parameters are:

max_chains — The maximum number of record chains allowed in each directory block.
DEFAULT = 11

split_val — The maximum average nonzero record chain length allowed in the *P* directory block before a split must occur.
DEFAULT = 1.0

overflow — The maximum average nonzero record chain length allowed in a non-*P* directory block before a split must occur in that block.
DEFAULT = 1.5

initial_m — The initial value for *M*, or the initial number of directories when loading the table (should be a power of 2).
DEFAULT = 2

max_dirs — The maximum number of directories allowed per open table (should be a power of 2).
DEFAULT = 2048

sup_dups — True to suppress duplicate record keys; false to allow duplicate record keys.
DEFAULT = TRUE

sort_chain — True if record chains are to be kept sorted; false if record chains are not to be kept sorted.
DEFAULT = TRUE

The code for loading a table named "table1" into a previously opened database is:

```
char      table_name[28];
PARM_TYPE new_parm;

strcpy(table_name, "table1");
if ((result = P_load(table_name, &new_parm)) != OK)
{
     printf("ERROR P_load returned %d\n", result);
}
```

4.8 Opening a Table

To open a table that has already been loaded, the command is *P_open(table_name, access_mode, &td)*, where the name of the table to be opened is passed in the parameter *table_name*. The access permission is passed in a variable of type ACCESS_TYPE, and the only permissible values are READ, WRITE, and READ_WRITE, which are defined in **pamm.h**. The third parameter to *P_open()* is the address of a table descriptor (*&td*) that is used for all successive table accesses for the opening process. The table descriptor is of type PAMM_TYPE and must be allocated before *P_open()* is called. For successful execution, each process that accesses a PAMM table must perform its own *P_open()* on that table. Upon successful completion, *P_open()* returns a valid table descriptor, which is used to access the table. The code for opening a table that was previously loaded is:

```
char            table_name[28];
ACCESS_TYPE     access_mode;
PAMM_TYPE       td;

strcpy(table_name, "table1");
access_mode = READ_WRITE;
if ((result = P_open(table_name,access_mode,&td)) != OK)
{
     printf("ERROR P_open returned %d\n", result);
}
```

4.9 Closing a Table

To close a table, the command is *P_close(&td)*, where *td* is a valid table descriptor of the table to be closed. When *P_close()* returns, the table descriptor is no longer valid. Further accesses with the table descriptor return an error message. *P_close()* does not unload the table but simply decrements its open count. The code for closing a table is:

```
PAMM_TYPE td;

if ((result = P_close(&td)) != OK)
{
     printf("ERROR P_close returned %d\n", result);
}
```

4.10 Saving a Table

To save an intermediate copy of a loaded table to disk, the command is *P_save(table_name, save_file_name)*. The maximum length of the table name is 27 characters, excluding file extension (".dat\0"), and the maximum length of the saved file name is 31 characters. It is recommended that the new saved file name not be the same as the table name plus file extension (".dat") because when the table is unloaded, it overwrites the intermediate copy. The code to save a table is:

```
char table_name[28];
char save_file_name[32];

strcpy(table_name, "table1");
strcpy(save_file_name, "new_file_name");
if ((result = P_save(table_name, save_file_name)) != OK)
{
     printf("ERROR P_save returned %d\n", result);
}
```

4.11 Unloading a Table

To unload a table that is no longer open, the command is *P_unload(table_name)*. If the table is still open by any process, *P_unload()* fails, immediately returning TABLE_STILL_OPEN. The code to unload a table is:

```
char table_name[28];

strcpy(table_name, "table1");
if ((result = P_unload(table_name)) != OK)
{
     printf("ERROR P_unload returned %d\n", result);
}
```

4.12 Deleting a Table

To delete a table, the command is *P_delete_table(table_name)*. This command deletes the data file *table_name*.dat and removes the information about the table and its attributes from the two data dictionaries. This command must be called before *P_open_DB()* or after *P_close_DB()* because it strictly alters the disk versions of the data dictionaries, not the PAMM structures. This operation is nonreversible. The code to delete a table is:

```
char table_name[28];

strcpy(table_name, "table1");
if ((result = P_delete_table(table_name)) != OK)
{
   printf("ERROR P_delete_table returned %d\n", result);
}
```

4.13 Inserting a Record

To insert a record into a table, the command is *P_insert(&td, record_ptr)*, where *td* is a table descriptor of type PAMM_TYPE and *record_ptr* is a pointer to the data record being inserted. Currently the record's key field must be the first four bytes (an integer) of the data record, however, this restriction may be removed in later versions of PAMM. *P_insert()* will return the value OK if the insertion completed successfully, NO_WRITE_PERM if the table was not opened with WRITE or READ_WRITE permission, or DUPLICATE_KEY if a record with that key is already in the table and duplicate suppression is turned off. The code to insert a record is:

```
PAMM_TYPE td;
char      *record_ptr;

/* data_record is defined and allocated elsewhere */
record_ptr = &data_record;
if ((result = P_insert(&td, record_ptr)) != OK)
{
     printf("ERROR P_insert returned %d\n", result);
}
```

4.14 Reading a Record

To read a record from a table, the command is *P_read(&td, key, record_ptr)*, where *td* is a table descriptor of type PAMM_TYPE, *key* is a 32-bit integer key value of the record to be retrieved, and *record_ptr* is a pointer to a previously allocated buffer where the retrieved record can be placed. *P_read()* will return OK if it found the record, TABLE_NOT_OPEN if the table was not open or the permissions were wrong, or RECORD_NOT_FOUND if the record was not found in the table. The code to insert a record is:

```
int       key;
char      *record_ptr;
PAMM_TYPE td;
```

```
/* data_record is defined and allocated elsewhere */
record_ptr = &data_record;
key = *((int *) record_ptr);
if ((result = P_read(&td, key, record_ptr)) != OK)
{
     printf("ERROR P_read returned %d\n", result);
}
```

4.15 Updating a Record

To update a record in a table, the command is *P_update(&td, record_ptr)*, where *td* is a table descriptor of type PAMM_TYPE and *record_ptr* is a pointer to a record whose contents will replace the record in the table with the same key field. The key of the record to be updated is found in the first four bytes of the record pointed to by *record_ptr*. If there is no record with that key in the table, an error message will be printed and the value RECORD_NOT_FOUND will be returned. The code to update a record is:

```
char      *record_ptr;
PAMM_TYPE td;

/* data_record is defined and allocated elsewhere */
record_ptr = &data_record;
if ((result = P_update(&td, record_ptr)) != OK)
{
     printf("ERROR P_update returned %d\n", result);
}
```

4.16 Deleting a Record

To delete a record from a table, the command is *P_delete(&td, key)*, where *td* is a table descriptor of type PAMM_TYPE and *key* is a four-byte integer whose value is the key of the record to be deleted. *P_delete()* removes the record from the table, but it does not release the memory back to the system for reuse until the database is closed (*P_close_DB()*). The code to delete a record is:

```
int       key;
char      *record_ptr;
PAMM_TYPE td;

/* data_record is defined and allocated elsewhere */
record_ptr = &data_record;
key = *((int *) record_ptr);
if ((result = P_delete(&td, key)) != OK)
{
     printf("ERROR P_delete returned %d\n", result);
}
```

4.17 Initiating a Database Session

As an example, we show a portion of a C program that opens a previously created database named "DB_name" and then loads two tables named "table1" and "table2" into PAMM structures. At this point, it copies 1000 records from "table1" into "table2" by forking 10 processes and letting each process copy a block of 100 records. Then it unloads both tables and closes the database. "table1" will have its PAMM structure parameters modified for different performance. The code is as follows:

```
int        result;
char       database_name[29];
char       table_name1[28];
char       table_name2[28];
PARM_TYPE  new_parm;

/* PAMM is initialized just once in a program, */
/* before any other PAMM calls are made. */
if ((result = P_init()) != OK)
{
     printf("ERROR P_init returned %d\n", result);
}

/* Open the  database */
/* The name of our example database is "DB_name" */
strcpy(database_name, "DB_name");
if ((result = P_open_DB(database_name)) != OK)
{
     printf("ERROR P_open_DB returned %d\n", result);
}

/* Load "table1" and change its PAMM table parameters */
strcpy(table_name1, "table1");
new_parm.max_chains = 512;
new_parm.split_val  = 3.01;
new_parm.overflow   = 5.01;
new_parm.initial_m  = 32;
new_parm.max_dirs   = 27;
new_parm.sup_dups   = TRUE;
new_parm.sort_chain = FALSE;

if ((result = P_load(table_name, &new_parm)) != OK)
{
     printf("ERROR P_load returned %d\n", result);
}

/* Load "table2" but don't alter PAMM table parameters*/
strcpy(table_name2, "table2");
if ((result = P_load(table_name, NULL)) != OK)
{
     printf("ERROR P_load returned %d\n", result);
}

/* Fork some processes to do concurrent database work */
```

```
/* and wait for them to finish */
for (i=0; i < 10; i++)
{
     if (fork() == 0)
     {
          task(i);
     }
}
for (i=0; i < N_tasks; i++)
{
     wait();
}

/* Unload the tables with their new or altered data */
if ((result = P_unload(table_name1)) != OK)
{
     printf("ERROR P_unload returned %d\n", result);
}
if ((result = P_unload(table_name2)) != OK)
{
     printf("ERROR P_unload returned %d\n", result);
}

/* And close the database */
if ((result = P_close_DB(database_name, FALSE)) != OK)
{
     printf("ERROR P_close_DB returned %d\n", result);
}
```

Each task opens both tables; "table1" is opened with read permission and "table2" is opened with write permission. Records are read from "table1" into a buffer named *buf*[] and then written into "table2" from that buffer. Record keys are computed by allocating each process a block of 100 records. Thus, process 1 copies records with keys in the range of 0 to 99, and process 2 copies records with keys in the range of 100 to 199, etc. Then both tables will be closed. The code for each task is:

```
task(no)
int no;
{
     int        key
     PAMM_TYPE td1;
     PAMM_TYPE td2;
     char       buf[1000];

     /* Open both tables */
     if ((result = P_open("table1", READ, &td1)) != OK)
     {
          printf("ERROR P_open returned %d\n", result);
     }
     if ((result = P_open("table2", WRITE, &td2)) != OK)
     {
          printf("ERROR P_open returned %d\n", result);
     }
```

```
        /* Copy 100 records from table1 to table2 */
        for (i=0; i < 100; i++)
        {
           key = (no * 100) + i;
           if ((result = P_read(&td1, key, buf)) != OK)
           {
              printf("ERROR P_read returned %d\n",result);
           }
           if ((result = P_insert(&td2, buf) != OK)
           {
              printf("ERROR P_insert returned %d\n",result);

           }
        }

        /* Close both tables */
        if ((result = P_close(&td1)) != OK)
        {
              printf("ERROR P_close returned %d\n", result);
        }
        if ((result = P_close(&td2)) != OK)
        {
              printf("ERROR P_close returned %d\n", result);
        }

}
```

5.0 DEBUGGING PAMM

In the unlikely occurrence that PAMM contains software bugs, debugging messages have been included throughout the PAMM source code. They are turned off by default but are easily activated by editing **pamm.h** and removing the character 'X' (capital X) from in front of any debug macros (they are all in one debug section). Each debug macro corresponds with a particular PAMM source code module (except for one macro: DEBUG, which is a catchall for some lower level routines) and are easily correlated. The application program and the PAMM must then be recompiled (PAMM using the make file). To turn the debug messages off again, place the 'X' in front of the debug macros that had been activated and recompile the source code. When they are turned off, they do not affect the performance of PAMM because they are not compiled into the code.

When the debugging messages are activated, they will be printed on the screen. If that is undesirable, they may be printed to a file named "PAMM_debug_messages" by editing **pamm.h** and removing the 'X' from in front of the macro "DEBUG_FILE_ON" and recompiling PAMM and the application source code. If the application is executed a second time, the contents of the debug messages file from the previous execution will be lost if not saved.

6.0. EVENT LOGGER

The GP1000 can log histories of events that occur during a program's execution. We have included this facility for monitoring and optimizing PAMM's performance. Macro calls are inserted into the code at specific locations to record the time that a process executes that command.

The time stamps are saved in a file that is later viewed on an X terminal by using the "gist" command. (See the Mach 1000 operating system manuals for using the event logger and gist.)

Events have been inserted into the PAMM source code for recording events. To turn this feature on, uncomment (or define) the macro ELOG in **pamm.h** and recompile (using the make file) the PAMM code and the application program. If events are desired in the application code, they may be added also. For each forked process that wants to log events, two macros must be called: TASK_START_UP and TASK_CLEAN_UP. Both macros can be found in **pamm.h**.

This feature is useful for finding performance bottlenecks in the application code. The event logging facilities are turned off by default and, when turned off, do not affect the performance of PAMM because they are not compiled into the code. When the event logging facilities are turned on, the overhead incurred is minimal, so performance should not be altered significantly.

7.0 References

[1] T. Rosenau and S. Jajodia, "Basic Database Operations on the Butterfly Parallel Processor: Experiment Results," NRL Memorandum Report 6173, Mar. 1988.

[2] BBN Advanced Computers Inc., *Butterfly GP1000 Overview*, November 10, 1988 (sales brochure).

[3] BBN Advanced Computers Inc., *Butterfly GP1000 Switch Tutorial*, 1989 (sales brochure).

[4] S. Pramanik and H. Davies, "Multi-Directory Hashing," Technical Report, Department of Computer Science, Michigan State University, Aug. 1988.

[5] S. Pramanik, "Key-Based, Distributed Locked RAM File System for Butterfly Machine Using Multi-Directory Hashing," Michigan State University, 1987.

[6] S. Pramanik and M. H. Kim, "Generalized Parallel Processing Models for Database Systems," 1988 International Conference on Parallel Processing, St. Charles, IL, Aug. 1988.

[7] S. Pramanik, C. Severance, and T. Rosenau, "A High Speed KDL-RAM File System for Parallel Computers," Proceedings of PARBASE-90, Miami Beach, Florida, Mar. 1990.

[8] C. Severance, T. Rosenau, and S. Pramanik, "A High Speed KDL-RAM File System for Parallel Computers," NRL Report 9259, Nov. 1989.

[9] T. Rosenau and M. El-Kadi, "The Design of a Parallel Access Main Memory (PAMM) DBMS on a Butterfly GP1000," NRL Report 9266 (in process).

[10] R. Rettberg and R. Thomas, "Contention is no Obstacle to Shared-Memory Multiprocessing," *Commun. ACM*, **29**(12), 1202-1212 (1986).

Appendix

PARALLEL ACCESS MAIN MEMORY (PAMM) User's Manual, Command Definitions

(Alphabetically Sorted)

PAMM Command Definitions

NAME

```
P_close
```

SYNOPSIS

```
#include <pamm.h>

ret = P_close(td);
int ret;
PAMM_TYPE *td;
```

DESCRIPTION

P_close() closes the table referenced by the table descriptor *td* decrementing the table's open count. It also invalidates *td* for later use in the calling program.

RETURN VALUES

OK	table was closed successfully
TABLE_NOT_OPEN	*td* is not a valid table descriptor

PAMM Command Definitions

NAME

```
P_close_DB
```

SYNOPSIS

```
#include <pamm.h>

ret = P_close_DB(db_name, wait);
int  ret;
char db_name[];
BOOLEAN wait;
```

DESCRIPTION

P_close_DB() closes the database called *db_name*. It unloads all the tables currently residing in the main memory data structure onto the disk and unloads the data dictionaries. The database cannot be closed while tables remain open. If one or more tables remain open, the *wait* parameter allows the calling program to specify whether *P_close_DB()* should wait for all tables to be closed before closing the database or fail and return an error message. If wait = TRUE (1), *P_close_DB()* will wait, but if wait = FALSE (0), *P_close_DB()* will fail, returning the code OPEN_TABLE_BUT_CANT_WAIT.

RETURN VALUES

OK	database was closed successfully
DB_CLOSE_FAILED	a fatal error occurred
OPEN_TABLE_BUT_CANT_WAIT	tables are still open and *wait* was set to FALSE (no wait)

PAMM Command Definitions

NAME

```
P_create
```

SYNOPSIS

```
#include <pamm.h>

ret = P_create(table_name);
int ret;
char table_name[];
```

DESCRIPTION

P_create() creates a table called *table_name*. A file called "*table_name*.scm" must exist in the current directory, which must be the schema file for the requested table, and it must be of the format described in NOTES below. *P_create()* adds the new table's information to the data dictionaries without loading the database into main memory. *P_create()* also creates an empty data file for the table named "*table_name*.dat". *P_create()* **must** be called before *P_open_DB()*, i.e., the database cannot be already loaded into main memory.

NOTES

The schema file format is as follows, where the reserved words are in bold and the user-supplied variables are in plain text:

database db_name
  attr attr1_name, **type** attr1_type, **key** 1
  attr attr2_name, **type** attr2_type, **key** 2
  attr attr3_name, **type** attr3_type, **key** 0
  .
  . *etc*
  .
  attr attrN_name, **type** attrN_type, **key** 0
end

Allowable values for **type** are:

TYPE	C TYPE	SIZE	DESCRIPTION
char	char	1 byte	8-bit character
short	short	2 bytes	16-bit, short integer
integer	int	4 bytes	32-bit integer
long	long	4 bytes	32-bit integer
float	float	4 bytes	32-bit, single-precision floating point
double	double	8 bytes	64-bit, double-precision floating point
string *N*	char[N]	N bytes	N-1 byte character string plus null terminator

In the current version, the values for **key** have no meaning and the first attribute must always be of type integer.

RETURN VALUES

OK	the table was created successfully
SCHEMA_NOT_FOUND	"*table_name*.scm" does not exist
SCHEMA_SYNTAX_ERROR	schema file is not in correct format
TABLE_ALREADY_EXISTS	table already created in the database
CREATE_FAILED	a fatal error occurred

PAMM Command Definitions

NAME

```
P_create_DB
```

SYNOPSIS

```
#include <pamm.h>

ret = P_create_DB(db_name);
int  ret;
char db_name[];
```

DESCRIPTION

P_create_DB() creates a new database with the name passed in *db_name*. If the database already exists, *P_create_DB()* fails and prints an error message. It creates and initializes the data dictionaries for the new database. It does not involve the main memory data structure.

RETURN VALUES

OK	database was created successfully
DB_ALREADY_EXISTS	dictionaries for *db_name* already exist
DB_CREATE_FAILED	a fatal error occurred

PAMM Command Definitions

NAME

```
P_delete
```

SYNOPSIS

```
#include <pamm.h>

ret = P_delete(td, key);
int ret;
PAMM_TYPE *td;
int key;
```

DESCRIPTION

P_delete() deletes a record with integer key, *key*, from the table described by *td*.

RETURN VALUES

OK	record was deleted successfully
NO_WRITE_PERM	table is not open with write permission
RECORD_NOT_FOUND	*key* does not exist in table *td*

PAMM Command Definitions

NAME

```
P_delete_table
```

SYNOPSIS

```
#include <pamm.h>

ret = P_delete_table(table_name);
int ret;
char table_name[];
```

DESCRIPTION

P_delete_table() deletes a table with the name *table_name*. It requires that a schema file of the correct format (see *P_create()*) exists called "*table_name*.scm". It deletes the given table's information from the dictionaries without loading the database into main memory. It also deletes the data file for the table, which is named "*table_name*.dat". It must be called before *P_open_DB()*, i.e., the database cannot be already loaded in main memory.

RETURN VALUES

OK	table was deleted successfully
SCHEMA_NOT_FOUND	"*table_name*.scm" does not exist
SCHEMA_SYNTAX_ERROR	"*table_name*.scm" is not in the correct format
TABLE_NOT_EXIST	*table_name* was not found in the dictionaries
DELETE_TABLE_FAILED	a fatal error occurred

PAMM Command Definitions

NAME

```
P_destroy_DB
```

SYNOPSIS

```
#include <pamm.h>

ret = P_destroy_DB(db_name);
int ret;
char db_name[];
```

DESCRIPTION

P_destroy_DB() destroys the database called *db_name*. It removes the data dictionaries and all data files belonging to its tables. It is not called on an open database.

RETURN VALUES

OK	database was destroyed successfully
DB_DID_NOT_EXIST	*db_name* does not exist
DESTROY_DB_FAILED	a fatal error occurred

PAMM Command Definitions

NAME

```
P_init
```

SYNOPSIS

```
#include <pamm.h>

ret = P_init();
int ret;
```

DESCRIPTION

P_init() allocates and initializes all of the global data structures that PAMM uses. It must be called before any other PAMM routine except *P_create_DB()*, *P_destroy_DB()*, *P_create()*, and *P_delete_table()*. It must be called by the parent task of all tasks that access the PAMM structures because this function sets the memory inheritance for all global memory. It returns the values OK if *P_init()* completed successfully or NOT_OK if there was an error.

RETURN VALUES

OK	PAMM was initialized successfully
NOT_OK	a fatal error occurred

PAMM Command Definitions

NAME

P_insert

SYNOPSIS

```
#include <pamm.h>

ret = P_insert(td, data_rec);
int ret;
PAMM_TYPE *td;
char data_rec[];
```

DESCRIPTION

P_insert() inserts a record, *data_rec*, into the table described by *td*. The key for the record is assumed to be an integer in the first four bytes of *data_rec*.

RETURN VALUES

OK	record was inserted successfully
NO_WRITE_PERM	table is not open or only open with read permission
DUPLICATE_KEY	key already exists and duplicates are suppressed

PAMM Command Definitions

NAME

P_load

SYNOPSIS

```
#include <pamm.h>

ret = P_load(table_name, new_parm);
int ret;
char table_name[];
PARM_TYPE *new_parm;
```

DESCRIPTION

P_load() loads the data from the table on disk, *table_name*.dat, into the main memory PAMM data structure. It then makes *table_name* available to be opened for use. The parameter *new_parm* specifies many different parameters for the control of the table. It is described in NOTES below. *P_load()* must be called before *P_open()*, even if the table is empty because it creates the table's directory structure.

NOTES

The type PARM_TYPE is defined in **pamm.h**. These parameters can be adjusted to improve the efficiency of the database.

```
typedef struct parm_type_x
{
     int        max_chains;
     float      split_val;
     float      overflow;
     int        initial_m;
     int        max_dirs;
     BOOLEAN    sup_dups;
     BOOLEAN    sort_chain;
} PARM_TYPE;
```

max_chains - The maximum number of record chains allowed in a directory block. DEFAULT = 11

split_val - The maximum average nonzero record chain length allowed in the *P* directory block before a split must occur. DEFAULT = 1.0

overflow - The maximum average nonzero record chain length allowed in a non-*P* directory block before a split must occur in that block. DEFAULT = 1.5

initial_m - The initial value for *M* at the opening of the database (should be a power of 2). DEFAULT = 2

max_dirs - The maximum number of directories allowed per open table (should be a power of 2). DEFAULT = 2048

sup_dups - TRUE to suppress duplicate keys, FALSE if duplicates keys are allowed. DEFAULT = TRUE

sort_chain - TRUE if record chains are to be sorted, FALSE if record chains are kept unsorted. DEFAULT = TRUE

RETURN VALUES

OK	*table_name* was loaded successfully
TABLE_NOT_EXIST	*table_name* does not exist in this database
TOO_MANY_TABLES	too many tables are currently open (20)
LOAD_FAILED	a fatal error occurred